Design by Contract

What it is and how ASCI might use it

An ASCI Verification and Validation Seminar

Presented by

David E. Peercy
Quality Engineering Department
12326
844-7965
depeerc@sandia.gov

Kent G. Budge
Computational Physics R&D
09231
284-3825
kgbudge@sandia.gov

Objectives

- Provide Summary of Design by Contract
  - A technique that may assist ASCI designers and V&V community in producing more reliable software codes
- Briefly Summarize an Implementation Instance
  - C++ language implementation
- Identify Possible ASCI Application Areas
  - How can we apply the concepts of DbC to ASCI?
- Answer Questions and Determine Interest Level
  - Issues
  - Possible code team interest

Overview
Design by Contract

- Overview of Design by Contract
  Bertrand Meyer One-Day Course, 27Sep99
  - Course Slides (PDF File)
  - Object-Oriented Software Construction, 2nd Edition 1997
    - Chapter 11, Design by Contract
  - Handout Papers (to be delivered)
  - Web Site: http://eiffel.com => Interactive Software Engineering
- Contract Mechanism
  - Assertions
  - Preconditions
  - Postconditions
  - Invariant
  - Abstract Data Type (ADT) and Classes

Course Key Questions

- ADVERTISED (was it discussed in course?)
  - What's Design by Contract beyond the buzzword? (yes)
  - How much of Design by Contract can be applied in Java and C++? (yes)
  - How much can you do in classical languages such as C? (yes)
  - What gains can you expect in terms of quality and productivity? (a little)
  - How can contracts be combined with component technologies such as COM/DCOM and CORBA? (a little)
  - How does Design by Contract fit with quality-enhancing standards such as ISO 9001 and the CMM? (no)
  - How can you ascertain the quality of software components? (yes)
  - How can developers produce useful documentation without huge extra work? (yes)
  - How does Design by Contract affect the software lifecycle and project management? (no)
  - What tools are available today to support Design by Contract? (a little)

Course Outline: Part 1

- PART 1: ISSUES (was it discussed in course?)
  - Software reliability (yes)
    - How important is it? Can we get away with
    - Good Enough Software? How does the industry cope with bugs and other reliability problems?
    - Components of reliability: correctness, robustness; role and limits of quality assurance; role and limits of formal methods.
  - Reliability techniques (a little)
    - typing, O-O structure, garbage collection, and others. Management-oriented approaches: ISO 9001, Capability Maturity Model (SEI)
  - Reliability and the software process (a little)
    - what is the role of each phase?
  - Reliability and the component revolution. (yes)

Course Outline: Part 2

- PART 2: PRINCIPLES (was it discussed in course?)
  - The theoretical basis (yes)
    - assertions and formal semantics.
  - The notion of contract (yes)
    - human contracts, software contracts. How far does the metaphor extend? What is special about software contracts?
  - Introducing contracts into software (yes)
    - preconditions, post-conditions, class invariants and others. How does this fit in an object-oriented software architecture? What's special about objects and contracts?

This part was the main focus of the course.

Course Outline: Part 3

- PART 3: APPLICATIONS (was it discussed in course?)
  - Contracts and documentation: how to produce good software documentation (and live to tell the tale) (yes)
  - Contracts and analysis: real developers do use bubbles! (no)
  - Contracts and implementation: killing the defects before they kill you (yes)
  - Contracts and debugging: rehabilitating the most shameful part of the business. (yes)
  - Contracts and testing: a systematic approach (no)
  - Contracts and quality assurance: a unifying concept. (yes)
  - Contracts and components: making reuse succeed. (yes)
  - Contracts and abnormal cases: a sound basis for exception handling. (yes)

Course Outline: Part 4

- PART 4: TOOLS (was it discussed in course?)
  - Programming languages and contracts (yes)
    - Ada, C++, Eiffel, Sather
    - Java and C++ extensions for contracts
  - Contracts and UML (yes)
    - the Object Constraint Language
  - Contracts and component technologies (no)
    - using Design by Contract to take the best advantage of Microsoft's COM and DCOM and the OMG's CORBA standard
  - Contracts and standards (no)
    - ISO 9001, CMM
  - Development environments (somewhat - Eiffel)
    - their support for contracts
  - A window on research - beyond current approaches (??)

Introduction & Overview

- Design By Contract: Scope
  - Methodological principles
    - Language- and tool-independent.
  - Applications
    - quality assurance
    - debugging, testing
    - documentation
    - exception handling, inheritance
  - Language and tool support:
    - Eiffel language (built-in); tools (EiffelBench, EiffelCase); Business Object Notation (BON) analysis & design method & notation
    - Can be partially emulated in C++ through macros; various proposed extensions for Java; extensions proposed for other languages.

Introduction & Overview

- Design By Contract: Scope
  - Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users).
  - This goal is the element's contract.
  - The contract of any software element should be explicit - Part of the software element itself.
  - *A NEW VIEW OF SOFTWARE CONSTRUCTION*
    - Constructing systems as structured collections of cooperating software elements — clients and suppliers — cooperating on the basis of clear definitions of obligations and benefits.
    - These definitions are the contracts.

Introduction & Overview

- Design By Contract: Component Context

  Components Provide

  - Data abstraction, classes, information hiding
    - to separate component implementation from component interfaces
  - Polymorphism and dynamic binding
    - to allow for dynamic adaptation of components to actual client needs
  - Inheritance and Genericity
    - for organizing components in rational hierarchies
  - Design by Contract
    - to make sure that components are properly specified and validated
    - to facilitate supportability and reliability

Introduction & Overview

- Properties Of Contracts
  - Binds two parties (or more): client, supplier
  - Is explicit (written): language
  - Specifies mutual obligations and benefits conditions
  - Usually maps obligation for one of the parties into benefit for the other, and conversely
  - Has no hidden clauses: obligations and benefits are those specified
  - Often relies, implicitly or explicitly, on general rules applicable to all contracts (laws, regulations, standard practices)

Introduction & Overview

- *Example Human Contract (FedX Delivery)*
  - Client: needs a package to be delivered to specific location by a specific time
    - Postcondition: package must be delivered by 10am next day
  - Supplier (FedX): is in the business of delivering packages to specific locations by specified times within some constraints
    - Precondition: if the Client can get the package to Supplier by 4pm on the current day and pay the requested fee
    - Postcondition: package will be delivered to specified location by 10am next day
  - Corporate Policy (class invariant): client is always treated with respect by FedX suppliers

- *Example Human Contract (FedX Delivery)*

	Obligations	Benefits
Client	(Satisfy precondition:) Bring package before 4 PM; pay fee.	(From postcondition:) Get package delivered by 10 AM next day. (From class invariant:) Feel like well-treated customer
Supplier	(Satisfy postcondition:) Deliver package by 10 AM next day. (Satisfy class invariant:) Treat client with respect	(From precondition:) Not required to do anything if package delivered after 4 PM, or fee not paid (From class invariant:) Client may return for more business

A natural question: what about the other obvious exception?

Introduction & Overview

- *Example Analysis Contract*

```
deferred class VAT inherit
                TANK
feature
  in_valve, out_valve: VALVE
  fill is
                -- Fill the vat
    require
      in_valve.open; out_valve.closed
    deferred
    ensure
      in_valve.closed; out_valve.closed; is_full
    end
  empty, is_full, is_empty, gauge, maximum,
  ---[other features]
invariant
  is_full = ((gauge >= 0.97*maximum) and (gauge <= 1.03*maximum))
end
```

Specified only
not implemented

Precondition

Postcondition

Class Invariant

Introduction & Overview

- *Example Analysis Contract*

	Obligations	Benefits
Client	**(Satisfy precondition:)** Make sure input valve is open, output valve is closed.	**(From postcondition:)** Get filled-up vat, with both valves closed.
Supplier	**(Satisfy postcondition:)** Fill the vat and close both valves. **(From class invariant:)** Vat must not be over or under filled.	**(From precondition:)** Simpler processing thanks to assumption that valves are in the proper initial position. **(From class invariant:)** No product waste due to overfill (cost/environment saving), or irritated client due to underfill (client return business).

Issues

- Quality Factors of Great Importance
  - Reliability
    - correctness
    - robustness
  - Supportability
    - reusability
    - extendability
    - portability
    - ...

Correctness: the ability of a software system to perform according to the specification, in cases defined by the specification.

Robustness: the ability of a software system to react in a reasonable manner to cases not covered by the specification.

Issues

- Software Quality
  - not principal concern of decision makers, many instances of non-quality: Ariane 5 rocket recent one
  - approaches to quality (technical)
    - "Test, test and retest"
    - Formal specification and verification: Z, B, VSE, OBJ, VDM,...
    - Partly formal: Design by Contract
    - Programming language support: strong typing, object technology,…
    - Style standards
  - approaches to quality (managerial)
    - CMM, ISO 9001
    - Buy from market leader
    - Metrics collection and application
    - Code reviews
  - approaches to quality (component)
    - Reuse, components, COTS, CBD, ...

Issues

- Terms to Denote Software Woes
  - error: a wrong decision made during the development/support of a software system
  - defect: a property of a software system that may cause the system to depart from its intended behavior
  - fault/failure: an event of a software system departing from its intended behavior during one of its executions
  - bug: synonymous with defect

Note: sometimes "fault" is synonymous with "defect" and "failure" is as defined; sometimes all terms are misused in relation to these definitions.

Assertion violation rules:

(1) a run-time assertion violation is the manifestation of a defect in the software
(2) a precondition violation is a manifestation of a defect in the client
(3) a postcondition violation is a manifestation of a defect in the supplier
(4) a class invariant violation is a manifestation of a defect in the supplier

Issues

- The Road Towards a Solution (is multi-faceted)
  - Component-based development
  - Formal or partially formal techniques (Design by Contract)
  - Object technology
  - Modern programming language techniques
  - Systematic testing
  - Open source
  - Systematic metrics collection and analysis
  - Management, engineering process

Principles

- Correctness in Software
  - Correctness is a relative notion: consistency of implementation vs specification -- assuming there is a specification
  - Basic notation: (P, Q: assertions, I.e., properties of the state of the computation. A: instructions).
- Correctness Formula
  - Hoare triple (after C.A.R. (“Tony”) Hoare -- formerly of Oxford University and now, as of recently, Microsoft Research.

$$\{P\}\ A\ \{Q\}$$

- Meaning of the Correctness Formula

> *Any execution of A started in a state satisfying P will terminate in a state satisfying Q*

Principles

- Example:

  {X>=9} X:=X+5 {X>=13}

- Weak and Strong Conditions

  Case 1: {False} A {...}

  Case 2: {...} A {True}

- Exercise
  - An ad is placed in the paper for a job. Preconditions for being qualified for the job, job applicant activities, and postcondition for successfully completing the job.
  - Would the applicant want very strong {P} or very weak {P}? What {P} would be best? Would Case 1 or Case 2 be better?

Principles

- Assertions, Preconditions, Postconditions, Invariants
  - Assertions are Boolean statements
  - "P" statements constitute the precondition assertions: expresses the constraints under which a routine will function properly
  - "Q" statements constitute the postcondition assertions: expresses properties of the state resulting from a routine's execution
  - Class Invariants
    - Preconditions and postconditions describe the properties of individual routines. There is also a need for expressing global properties of the instances of a class, which must be preserved by all routines. Such properties will make up the class invariant, capturing the deeper semantic properties and integrity constraints characterizing a class.
    - Human contracts example: general clauses, regulations that apply to all contracts within certain categories such as fire, electrical codes, etc to house construction contracts

Principles

- *Example Analysis Contract (Again)*

```
deferred class VAT inherit
                TANK
feature
  in_valve, out_valve: VALVE
  fill is
                -- Fill the vat
    require
      in_valve.open; out_valve.closed
    deferred
    ensure
      in_valve.closed; out_valve.closed; is_full
    end
  empty, is_full, is_empty, gauge, maximum,
  ---[other features]
invariant
  is_full = ((gauge >= 0.97*maximum) and (gauge <= 1.03*maximum))
end
```

Specified only
not implemented

Precondition

Postcondition

Class Invariant

Principles

- "Language" Summary (Design by Contract)
  - precondition
    - require {requirements that must be satisfied when a routine is called}
  - postcondition
    - ensure {requirements that must be satisfied when a routine ends}
  - class invariants
    - invariant {requirements all class routine pre/post conditions must satisfy}
  - exception handling
    - rescue {admit defeat and raise exception to caller}
    - retry {gallantly try a new strategy or retry the old one, then rescue}
  - other constructs
    - check {can be used by client to check precondition assertions}
    - old {allows the retention of input values for comparison with output results, for example in postconditions}
    - ...

Principles

```
deferred class VAT inherit
                TANK

feature
  in_valve, out_valve: VALVE
  fill is
                -- Fill the vat
    require
      in_valve.open; out_valve.closed

    deferred      -- main body part deferred

    ensure
      in_valve.closed; out_valve.closed; is_full

    rescue
      -- one retry then terminate
      retry

  end -- fill

  empty, is_full, is_empty, gauge, maximum,
  ---[other features]

invariant
  is_full = ((gauge >= 0.97*maximum) and (gauge <= 1.03*maximum))
end
```

Precondition

Exception Handling

Postcondition

Class Invariant

Principles

- Preconditions (restrictions)
  - Accessibility
    - Preconditions can only be written in terms of methods and data accessible to the client; cannot reference protected/private members of the class. Not “fair” for supplier to “hide” precondition info!
  - Side Effects
    - No assertion of any kind may introduce side effects.
  - Inheritance
    - If virtual methods of a base class are overridden by a derived class, the overriding method may not strengthen preconditions, otherwise clients would not know what preconditions have to be met.
  - Redundancy Principle
    - Under no circumstances shall the body of a routine ever test for the routine’s precondition.

Principles

- Postconditions (restrictions)
  - Accessibility (same as preconditions)
    - Postconditions can only be written in terms of methods and data accessible to the client since client expects the postconditions to hold after execution; Not “fair” for client to “hide” postcondition info!
  - Side Effects (same as preconditions)
    - No assertion of any kind may introduce side effects.
  - Inheritance
    - If virtual methods of a base class are overridden by a derived class, the overriding method may not weaken the postconditions, otherwise clients would not know what postconditions have to be met.

Principles

- Class Invariants
  - Definition
    - An invariant for a class C is a set of assertions such that every instance of C will satisfy at all "stable" times. Stable times are those in which the instance is in an observable state (creation, before/after routine call)
  - Invariant Rule (class C, invariant I)
    - Creation of class C with satisfied preconditions yields a state satisfying I; and
    - Any exported routine of class C, when applied to arguments and a state satisfying both I and the routine's precondition, yields a state satisfying I. {INV and pre} body {INV and post} in Hoare's notation
  - Variations of the Invariant
    - Invariants may be modified within the body of a class/routine as long as the invariant rule is satisfied
    - *Invariant Rule:* the invariant of a class automatically includes the invariant clauses from all its parents, "and"-ed

Principles

- Exception Handling
  - Exception: a run-time event that may cause a routine call to fail; a failure of a routine causes an exception in its caller
  - Two legitimate responses to an exception that occurs during the execution of a routine:
    - Retrying: attempt to change the conditions that led to the exception and to execute the routine again from the start
    - Failure (also known as organized panic): clean up the environment, terminate the call, and report failure to the caller.
  - Basic syntax

```
routine is
        require
          precondition
        local
          …local entity declarations
        do
          body
        ensure
          postcondition
        rescue
          rescue_clause (may contain retry clause)
        end
```

Principles

- Other Constructs: Check
  - Check instruction serves to express the software writer's conviction that a certain property will be satisfied at certain stages of the computation.
  - Syntax

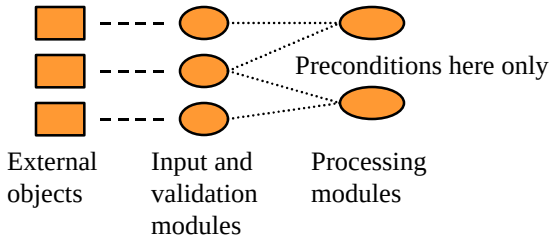
check
assertion_clause$_1$
assertion_clause$_2$
assertion_clause$_3$
---
assertion_clause$_n$
end

Applications

- What are Software Contracts Good For?
  - Writing correct software
    - analysis, design, implementation, maintenance, reengineering
  - Documentation
    - the “short” form of a class
  - Effective reuse
  - Controlling inheritance
  - Preserving the work of the best developers
  - Quality assurance, testing, debugging
    - especially in connection with the use of libraries
  - Exception handling

Applications

- What Software Contracts are NOT FOR?
  - Input Data Checking /Defensive Programming
    - should have specific modules/routines that check input data for correctness and, as appropriate, ensure the preconditions of any class routines that are subsequently called



  - Special Case Checking
    - should just use the typical control structures {if..then..else}
  - Human Interface Checks
    - there is no way to guarantee through software contracts that a human will not enter incorrect data, make sequence errors, etc.
    - however, the “commentary” statements of preconditions and capture of the preconditions for use by humans is still of value, and can be thought of as an extended use of the Design by Contract concept

Applications

- Example Project (ISE)
  - HP Laser printer software: 1997-1998
  - Embedded system development: software runs on chip in printer
  - Host development environment: VxWorks operating system
  - Size: 800,000 line of legacy C code
  - Process Steps:
    - Introduced Design by Contract in C and C++ through macros
    - Eiffel environment and language support introduced later, primarily because of memory management requirements; C calls Eiffel via existing CECIL library
  - Results:
    - Decreased error rates in the elements built with Design by Contract
    - Several major errors found in the legacy C code
    - Found bug in chip

Applications

- Contracts as a Safeguard for Software Evolution
  - ***Situation:*** company has a small group of hard guns who are responsible for the core job
  - ***Later:*** other engineers come in, and because they don't immediately understand the solution they start hacking it, and in the process destroy it
  - ***Consequence:*** quality of the code base is degraded to the level of the work of those who are not as good
  - ***Prevention:*** DbC addresses this issue by having the original designers build a white-box framework/scaffold into which implementations are plugged, with contracts that specify the vision

Applications

- Contracts as a Supportability Mechanism
  - Defect correction
    - defects are better isolated so correction requires less understanding and less testing of the corrected software
  - Enhancements
    - characteristics are built into software that make it easier to understand and change software without making mistakes, and to scale up software to handle enhanced capabilities and previously unknown applications
  - Adaptation
    - ensures confidence in the reuse of components in new applications and in new/adapted environments

Tools

- Language Methods
  - Eiffel: built-in
  - Contract extensions for: Ada 83, Smalltalk, C++, Java, Python, …
- Analysis and Design Methods/Tools
  - Business Object Notation (BON): build-in
  - UML extension: Object Constraint Language (OCL)

Tools

- Business Object Notation (BON)
  - BON provides a clear notation and methodological guidelines for high-level analysis and design: key concepts
    - seamlessness, reversibility and software contracting
  - Well-defined set of conventions; supports semantics (contracts, ...), not just structure
  - Mechanisms for systematic development; supports Design by Contract
  - Textual as well as graphical variants. Three views: graphics(bubbles&arrows!), tables, formal text (Eiffel-like).
  - Meant for use with software tools (EiffelCase)
- Scales up: abstraction and grouping facilities: classes, clusters,entire systems
- Reference:Kim Waldén and Jean-Marc Nerson, Prentice Hall, 1995

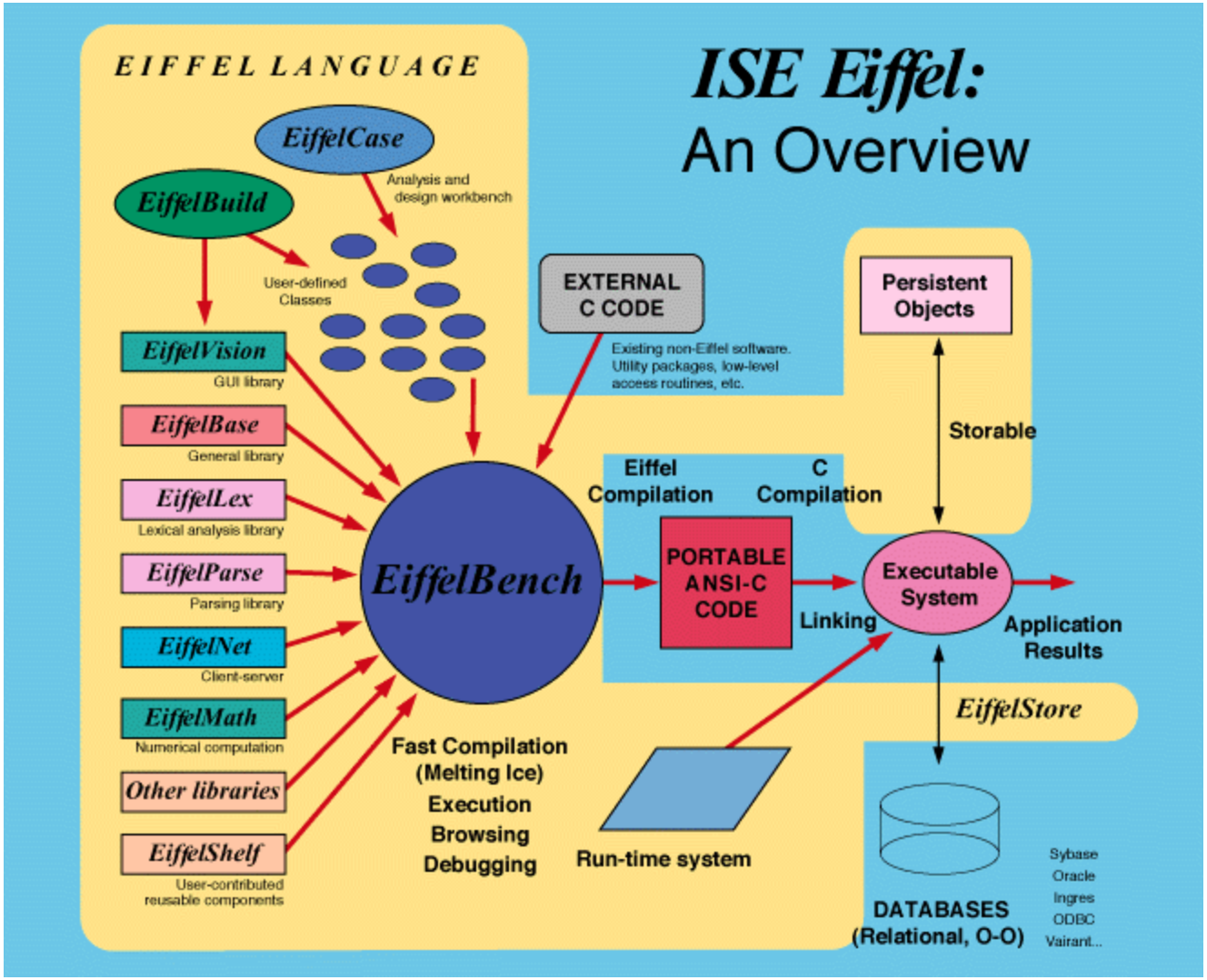
Tools

- EiffelCase Tool Set and Reengineering Library
  - Components of ISE Eiffel include:
    - **EiffelBench:** visual workbench for object-oriented software construction; automatic documentation tools; visual debugging.
    - **EiffelBase:** library of fundamental data structures and algorithms
    - **EiffelCase:** analysis and design workbench
    - **Embedded Eiffel:** environment adapted to the needs of embedded and real-time applications.
    - **EiffelCOM:** interoperability library using COM, OLE, ActiveX.
    - **EiffelCORBA:** interoperability library using CORBA.
    - **EiffelMath:** numeric/scientific computation on platforms supporting the NAG C library
    - **EiffelNet:** client-server & three-tier architectures: exchanging objects and object structures over a network

Tools

- EiffelCase Tool Set and Reengineering Library (cont)
  - **EiffelLex:** lexical analysis based on finite automata of various kinds
  - **EiffelParse:** object-oriented parsing mechanisms
  - **EiffelStore**: principal interface between Eiffel and external database management systems, relational or object-oriented
  - **EiffelWeb:** web form processing; uses Eiffel to write CGI scripts
  - **EiffelBuild**: application generator and graphical user interface builder
  - **DLE:** gives Eiffel developers the ability to integrate new classes into their systems at run time
  - **EiffelVision:** platform-independent graphical and graphical user interface (GUI) library; includes Windows Eiffel Library (WEL), and Motif Eiffel Library (MEL)
  - **Eiffel Resource Bench:** enables use of a Windows GUI builder (resource editor) to define the interface of an Eiffel application, through WEL
  - **EiffelThreads:** thread library providing multithreading
  - **Legacy++ :** C++ class wrapper: re-engineer C++ applications, wrapping them into Eiffel classes

Tools


EIFFEL LANGUAGE
ISE Eiffel:
An Overview
EiffelCase
Analysis and design workbench
EiffelBuild
User-defined Classes
EiffelVision
GUI library
EiffelBase
General library
EiffelLex
Lexical analysis library
EiffelParse
Parsing library
EiffelNet
Client-server
EiffelMath
Numerical computation
Other libraries
EiffelShelf
User-contributed reusable components
EiffelBench
Fast Compilation
(Melting Ice)
Execution
Browsing
Debugging
EXTERNAL C CODE
Existing non-Eiffel software. Utility packages, low-level access routines, etc.
Persistent Objects
Storable
Eiffel Compilation
C Compilation
PORTABLE ANSI-C CODE
Linking
Executable System
Application Results
EiffelStore
Run-time system
DATABASES (Relational, O-O)
Sybase
Oracle
Ingres
ODBC
Vairant...

Tools

- Applying Design by Contract in Non-Eiffel Environment
  - Basic Step
    - use standardized comments, or graphical annotations, corresponding to require, ensure, invariant clauses
  - In programming languages
    - macros: avoids the trouble of preprocessors, but invariants are more difficult to handle than preconditions and postconditions
    - preprocessor
  - Difficulties
    - contract inheritance
    - “short”-like tools
    - link with exception mechanism

Tools

- Design by Contract in C and C++
  - GNU Nana: improved support for contracts and logging in C and C++
    - P.J. Maker, Australia, see: http://www.cs.ntu.edu.au/homepages/pjm/nana-home/
    - Set of C++ macros and commands for gdb debugger. Replaces assert.h. Validated only with GCC
    - *"Support existed in earlier versions of Nana for the GNU Ada compiler. We may add support for Ada and FORTRAN in the future if anyone is interested."*
    - Support for quantifiers (Forall, Exists, Exists1) corresponding to iterations on the STL (C++ Standard Template Library).
    - Support for time-related contracts ("Function must execute in less than 1000 cycles")
  - See Kent Budge presentation on C++ example

Tools

- Design by Contract in Java
  - OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to “lack of time”.
  - “No assertions” is currently #4 on the Java users’ bug list. Several different proposals.
  - iContract
    - iContract, the Java Design by Contract Tool, TOOLS USA 1998, IEEE Computer Press, pages 295-307.
      - Java preprocessor. Assertions are embedded in special comment tags, so iContract code remains valid Java code in case the preprocessor is not available.
      - Support for Object Constraint Language mechanisms.
      - Support for assertion inheritance.
  - JASS (JAWA)
    - Preprocessor. Also adds Eiffel-like exception handling
    - http://theoretica.Informatik.Uni-Oldenburg.DE/~jawa/doc.engl.html

Tools

- The Object Constraint Language (OCL)
  - Designed by IBM and other companies as an addition to UML.
  - Includes support for:
    - Invariants, preconditions, postconditions
    - Guards
    - Predefined types and collection types
    - Associations
    - Collection operations: ForAll, Exists, Iterate
  - Not directly intended for execution.
  - Reference
    - http://www-4.ibm.com/software/ad/standards/ocl.html/

Tools

- The Object Constraint Language (OCL)
  - OCL is the expression language for the Unified Modeling Language (UML) ; has the characteristics of an expression, modeling, and formal languages
  - Expression language
    - OCL is a pure expression language; guaranteed to be without side effect
    - OCL expression can specify a state change; all values for all objects, including all links, will not change; evaluated OCL expression simply delivers a value
  - Modeling language
    - OCL is a modeling language, not a programming language; can't write program logic or flow-control in OCL; cannot invoke processes or activate non-query operations within OCL; not everything in it can be directly executable
    - All implementation issues are out of scope and cannot be expressed in OCL
  - Formal language
    - OCL is a formal language where all constructs have a formally defined meaning
    - Specification of OCL is part of the UML specification
    - Available from IBM at http://www.omg.org
    - OCL is not intended to replace existing formal languages, like VDM, Z

Tools

- The Trusted Components Initiative
- Initiated by:
  - Monash University (Melbourne)
  - Interactive Software Engineering
  - Univ. of Brighton, IRISA (France) and other institutions
- Reference
  - http://www.trusted-components.org

MISSON

Develop the infrastructure for enabling the software industry to transform itself into a discipline based on quality reusable components.

Implementation Instance
C++ Language

- Fundamentals of Design by Contract for C++
- C++ Language Constructs
  - Preconditions
    - C++ Structure
    - Rules for Preconditions
  - Postconditions
    - C++ Structure
    - Rules for Postconditions
  - Class Structure
    - C++ Structure
    - Invariants
    - Inheritance
- Conclusions

Opportunities
ASCI Application

- Requirements Analysis/Specification
  - Traceability: Customer - Software requirements contracts
- Design Analysis/Specification
  - Traceability: Requirements - Design contract specifications
  - Static Reviews and Dynamic Test: assertion violations
- Code Implementation
  - Traceability: Design - Code contracts
  - Static Reviews and Unit, Integration Testing Checks
- Verification and Validation
  - Static Reviews: check all contracts for validity
  - Internal Testing: turn on all assertions; localize debugging
  - Independent Testing: model contract validation
  - Planning: management to development contract verification

Opportunities
ASCI Application

- Requirements Analysis/Specification
  - Traceability: Customer - Software requirements contracts
- Design by Contract Structure
  - Client: customer/user
  - Supplier: code architect
  - Language Structures: natural language
    - preconditions
      - user provides: ...
    - postconditions
      - code returns: …
  - Requirements Tool Base
    - May be possible to use Unified Process/UML methods and tools or the Business Object Notation (BON) tools

Opportunities
ASCI Application

- Design Analysis/Specification
  - Traceability: Requirements - Design contract specifications
  - Static Reviews and Dynamic Test: assertion violations
- Design by Contract Structure
  - Client: Abstract Data Types/Classes/Features/Clusters
  - Supplier: Abstract Data Types/Classes/Features/Clusters
  - Language Structures:
    - can use the DbC assertion structures or specific language macros
    - evolve the system architecture design from ADTs to Classes (with deferred implementation)
    - could use Unified Process to derive use cases and scenario views from which DbC assertions and ADT/Classes etc could be derived
  - Design Tool Base
    - May be possible to use Unified Process/UML methods and tools or the Business Object Notation (BON) tools

Opportunities
ASCI Application

- Code Implementation
  - Traceability: Design - Code contracts
  - Static Reviews and Unit, Integration Testing Checks
- Design by Contract Structure
  - Client: Clusters/Classes/Features/Routines
  - Supplier: Clusters/Classes/Features/Routines
  - Language Structures:
    - can use the DbC assertion structures or specific language macros
    - evolve the implementation from Clusters/Classes (with deferred implementation) to actual implementations
    - check assertions through V&V activity: static reviews, testing
- Code Implementation Environment
  - May be possible to use Eiffel tool bench or other language (e.g., C++) checking tools to support implementation assertion checking

Opportunities
ASCI Application

- Verification and Validation
  - Static Reviews: check all contracts for validity
  - Internal Testing: turn on all assertions; localize debugging
  - Independent Testing: model contract validation
  - Planning: management to development contract verification
- Design by Contract Structure
  - Client: Customer/user
  - Supplier: Software code
  - Language Structures:
    - For requirements, design, implementation, testing use the embedded DbC assertion structures
    - For planning, use natural language structures
  - Common V&V Environment
    - Would be very useful to have a common V&V environment within which contract verification and validation could be controlled

Discussion Q&A

- Design by Contract
  - *Concepts?*
  - *Application?*
  - *Further References?*
- ASCI Interest and Application
  - Code Team Interest?
  - Next Steps?
    - On-going project within code implementation (Alegra, ??)
    - Static reviews, software inspections (Alegra, ??)
    - V&V Code Team Plans review process (planned pilot project)
    - Requirements analysis? (several code teams indicated interest)

References

- Meyers, Bertrand, “Object-Oriented Software Construction,” 2nd edition, Prentice Hall, NJ, 1997.
- Eiffel Website: http://eiffel.com
- Course Slides: see Tim Trucano, Kent Budge for a copy
- Course Papers: will make these available when received

Contacts

- SNL
  - Kent Budge, Tim Trucano, Dave Peercy
- LLNL

  Patrick J. 'Pat' Miller , mailcode: L-038
  phone: 925-423-0309 fax: 925-423-9208
  email: miller35@llnl.gov or patmiller@llnl.gov
  location: B111 R726, Scientific Computing Applications
- LANL
  - ??
- Bertrand Meyer ISE & Monash University

  Interactive Software Engineering
  ISE Building, 270 Storke Road
  Santa Barbara, CA 93117 USA
  Telephone 805-685-1006
  Fax 805-685-6869
  E-mail info@eiffel.com
  http://tools.com